ISOVALENT

Detection and Blocking with BPF via YAML

Kev Sheldrake

Senior Security Software Engineer, Isovalent

ISOVALENT

~~@kevsecurity / @kevsheldrake~~

kevin.sheldrake@isovalent.com /
kev@headhacking.com

- → MEng (University of York)
- → BAE SYSTEMS (software engineer, sysadmin, security engineer)
- → VEGA (security consultant)
- → Electric Cat (penetration tester, cat herder, money lender)
- → Head Hacking (hypnotist, magician)
- → BT (penetration tester, reverse engineer, security researcher)
- → Microsoft / Sysinternals (Linux security specialist)
- → Isovalent (senior security software engineer)

www.cosmic-pancakes.com

Cosmic Pancakes!

HOME ABOUT BASICS RESOURCES BLOG CONTACT

GREETINGS, HYPNO-SEEKERS.

And welcome to Cosmic Pancakes! Our quest is to understand the weird and wonderful world of hypnosis.

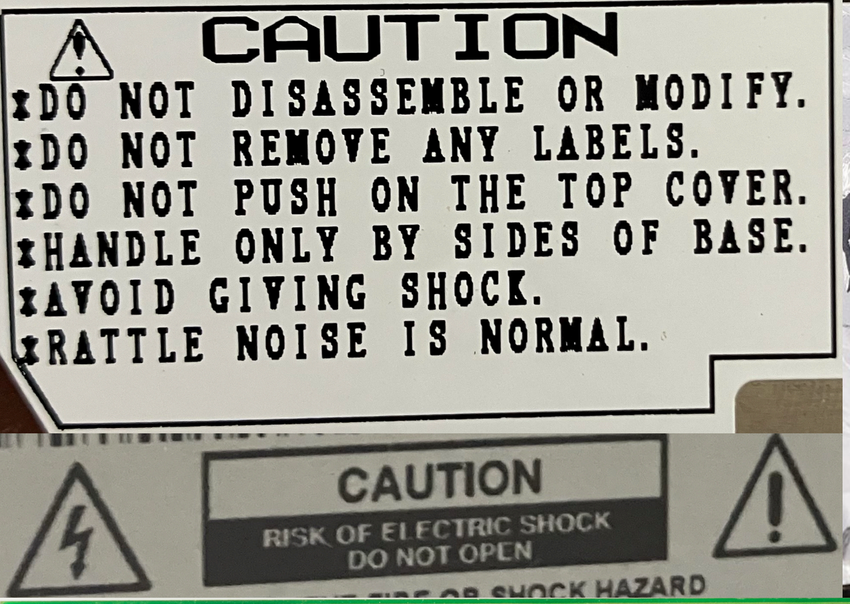
ISOVALENT

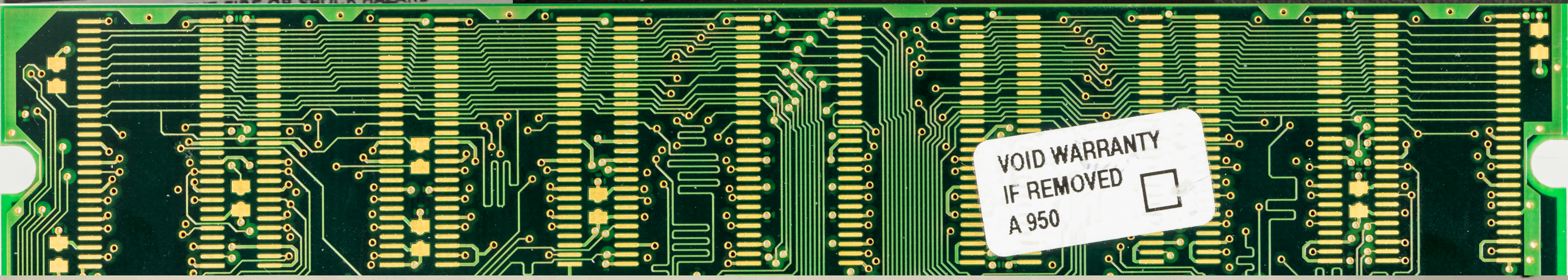
Overview

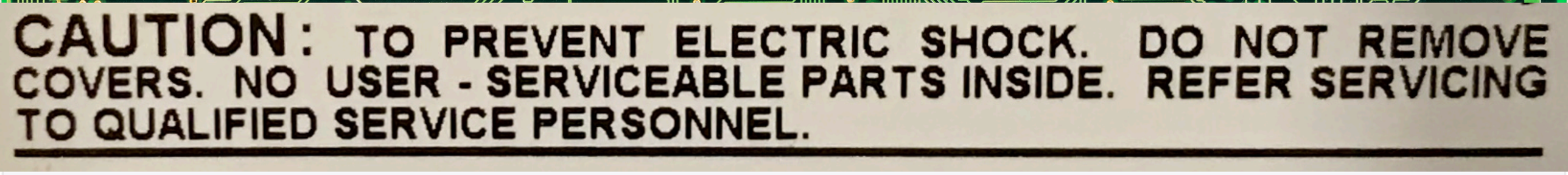
- Logging, past and future
- Summary of BPF: what it is; why it's significant; why it's traditionally difficult to use
- How Tetragon saves the day, with YAML! – Hurrah!
- Bundled items
- Examples
- Future plans

ISOVALENT

Logging







ISOVALENT

RAM and “Delicate Equipment” images credited to Raimond Spekking. “No User Servicable Parts” image credited to Wikimedia user: Vogler

ISOVALENT

Logging

- Off-the-shelf logging and detection systems:
  - Limited to what they think you need
  - Can’t predict the future
  - Don’t let you tinker beyond basic configuration

- Don’t *install* a logging system. *Build your own!*

ISOVALENT

Moare BPF?

Didn’t you do this already?

ISOVALENT

BPF summary (for those that missed it)

- “What is eBPF And Why Should You Care?”
  - BT Snoopcon, 44Con, OWASP South West
  - https://tinyurl.com/kev-what-is-ebpf
- BPF is POWERFUL
- BPF is (usually) DIY
- BPF is DIFFICULT

ISOVALENT

BPF is POWERFUL

- Hook (almost) any syscall, tracepoint, kernel function and network operation
- Run your own C code* – in the kernel – when the hook is reached (*caveats apply)
- Gather state from kernel memory and task struct
- Store, report, divert, kill, and more

ISOVALENT

BPF is (usually) DIY

- It's basically an API
- Most off-the-shelf BPF solutions limit how you can use it / what you can see or do; similar to:
  - kaudit-based solutions
  - kernel module-based solutions
- To get the most of out it, you need to build it yourself
  - Total flexibility over where to hook, what to report

ISOVALENT

BPF is DIFFICULT

- Write your hooks in C with inline ASM
  - Although support for Rust now
- Unfamiliar ecosystem
  - Maps, small stack, limited size, limited complexity, helpers, no native function calling
- Verifier *nightmares*

ISOVALENT

Instead

Why not make a generic eBPF tool that's really flexible and easy to configure?

ISOVALENT

Tetragon

Network & Security Observability &
Runtime Enforcement

Open Source at GitHub/Cilium/Tetragon

ISOVALENT

Generic

- All the code hidden from view
  - But is open source, so feel free to extend it :D
- Built to handle hooking
- Knows about data types:
  - Strings, character buffers, file descriptors, socks, SKBs, etc
- Can copy buffers on hook return

ISOVALENT

Flexible

- Match values in-kernel:
  - PIDs, binaries, arguments, namespaces, capabilities, return values
- Follow file descriptors and track sockets
- Report what you care about
- Outputs to gRPC, but rotating JSON logs optional
- Can trigger web and DNS canaries

ISOVALENT

Easy to configure

- Define tracing policy in YAML.
- Specify:
  - Where to hook
  - Which arguments are interesting (inc type)
  - The conditions you want to observe
  - The actions to take on a match

ISOVALENT

Tetragon Use Cases

- Logging system:
  - Replace kaudit/auditd/etc
  - Go beyond syscalls and log anything and everything
- Detection and response:
  - Block functionality
  - Kill processes from within the kernel
- Better strace:
  - Steal keys and credentials as they are used
- Debug kernel bugs

ISOVALENT

What's in the box?

ISOVALENT

Process management

- Logs all process executions and exits
- Stored in cache
  - Existing processes discovered on start up
- Used to annotate all other events
  - PID -> binary, arguments, CWD, start time, exec ID, parent exec ID, pod, namespace

ISOVALENT

Process execution


```
{
  "process_exec": {
    "process": {
      "exec_id": "OjI4MTQyNjQ1MzUzNDY4MjE6MzgyODc4Mw==",
      "pid": 3828783,
      "uid": 1000,
      "cwd": "/home/kev",
      "binary": "/usr/bin/ls",
      "arguments": "--color=auto -l",
      "flags": "execve clone",
      "start_time": "2023-06-14T15:55:21.942743693Z",
      "auid": 1000,
      "parent_exec_id":
"OjIwNzM0MDMzMzAwMDAwMDA6MTE3NDQwMg==",
      "tid": 3828783
    },
    "parent": {
      "exec_id":
"OjIwNzM0MDMzMzAwMDAwMDA6MTE3NDQwMg==",
      "pid": 1174402,
      "uid": 1000,
      "cwd": "/home/kev",
      "binary": "/usr/bin/bash",
      "flags": "procFS auid",
      "start_time": "2023-01-05T16:01:17.565437193Z",
      "auid": 1000,
      "parent_exec_id": "OjEwNjczNTAwMDAwMDA6NDg1Mw==",
      "refcnt": 3,
      "tid": 1174402
    }
  },
  "time": "2023-06-14T15:55:21.942742054Z"
}
```

ISOVALENT

Process exit


```
{
  "process_exit": {
    "process": {
      "exec_id": "OjI4MTQyNjQ1MzUzNDY4MjE6MzgyODc4Mw==",
      "pid": 3828783,
      "uid": 1000,
      "cwd": "/home/kev",
      "binary": "/usr/bin/ls",
      "arguments": "--color=auto -l",
      "flags": "execve clone",
      "start_time": "2023-06-14T15:55:21.942743693Z",
      "auid": 1000,
      "parent_exec_id":
"OjIwNzM0MDMzMzAwMDAwMDA6MTE3NDQwMg==",
      "tid": 3828783
    },
    "parent": {
      "exec_id":
"OjIwNzM0MDMzMzAwMDAwMDA6MTE3NDQwMg==",
      "pid": 1174402,
      "uid": 1000,
      "cwd": "/home/kev",
      "binary": "/usr/bin/bash",
      "flags": "procFS auid",
      "start_time": "2023-01-05T16:01:17.565437193Z",
      "auid": 1000,
      "parent_exec_id": "OjEwNjczNTAwMDAwMDA6NDg1Mw==",
      "refcnt": 1,
      "tid": 1174402
    },
    "status": 2,
    "time": "2023-06-14T15:55:21.944627712Z"
  },
  "time": "2023-06-14T15:55:21.944625907Z"
}
```

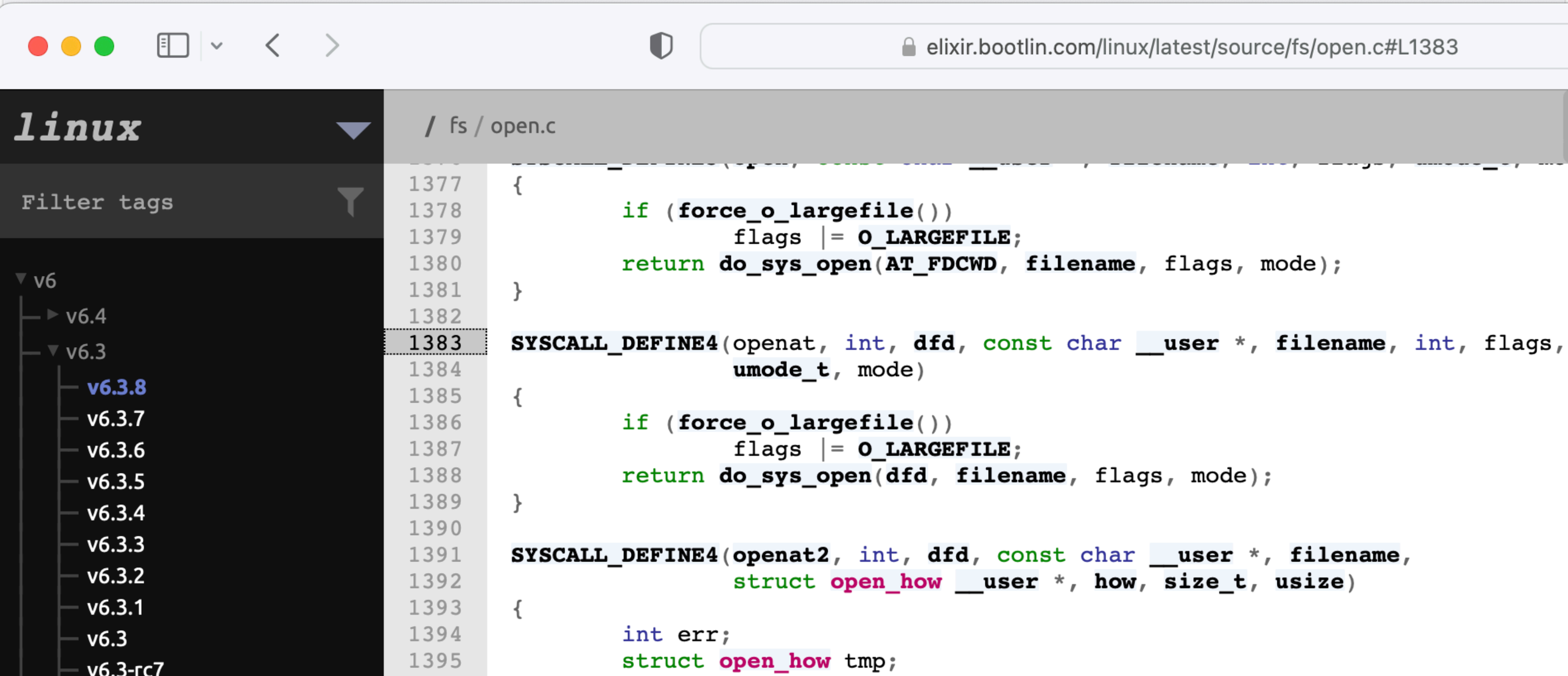
ISOVALENT

Tracing

- Support for hooking kprobes and tracepoints
- Documentation
- Examples
- Slack channel
- Continuous development
- *Community*

ISOVALENT

Where do we find hooks?



ISOVALENT

Where do we find hooks?

- elixir.bootlin.com
- Has all versions of vanilla kernel
- Search for:
  - “sys_[SYSCALL]”
  - “do_sys_[SYSCALL]”
- Follow the breadcrumbs
- Alternatively, grep the source code locally

ISOVALENT

Examples

ISOVALENT

Monitor clock_settime syscall

- sys_clock_settime(clockid_t which_clock, const struct kernel_timespec __user *tp)

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "sys-clock-settime"
spec:
  kprobes:
 - call: "sys_clock_settime"
    syscall: true
    args:
    - index: 0
      type: int
```

```
selectors:
- matchPIDs:
  - operator: NotIn
    followForks: true
    isNamespacePID: true
    values:
    - 1
  - operator: NotIn
    followForks: true
    isNamespacePID: true
    values:
    - 0
```

ISOVALENT

Monitor BPF syscall via kprobes

- int bpf_check(struct bpf_prog **prog, union bpf_attr *attr, bpfptr_t uattr) // program loads
- int security_perf_event_alloc(struct perf_event *event) // first step of kprobe attach
- int security_bpf_map_alloc(struct bpf_map *map) // called during BPF map create

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
 name: "bpf"
spec:
 kprobes:

- call: "bpf_check"
   syscall: false
   args:
   - index: 1
     type: “bpf_attr"

- call: "security_perf_event_alloc"
   syscall: false
   args:
   - index: 0
     type: "perf_event"

- call: "security_bpf_map_alloc"
   syscall: false
   args:
   - index: 0
     type: "bpf_map"
```

ISOVALENT

Detection vs alerting vs blocking vs killing

- Detection is great
- Alerting is cool
  - Thinkst Canaries (get URL or DNS look up)
- Blocking is good
  - Cancel function and specify return value
- Killing is awesome
  - Block and then send SIGKILL to process

ISOVALENT

Detection vs alerting vs blocking vs killing

- int symlinkat(const char *oldpath, int newdirfd, const char *newpath);

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "sys-symlink-passwd"
spec:
  kprobes:
  - call: "sys_symlinkat"
    syscall: true
    args:
    - index: 0
      type: "string"
    - index: 1
      type: "int"
    - index: 2
      type: "string"
```

```
selectors:
- matchArgs:
  - index: 0
    operator: "Postfix"
    values:
    - "passwd\0"
  matchActions:
  - action: DnsLookup
    argFqdn: abc.def.o3n.io
```

ISOVALENT

Detection vs alerting vs blocking vs killing

- int symlinkat(const char *oldpath, int newdirfd, const char *newpath);

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "sys-symlink-passwd"
spec:
  kprobes:
  - call: "sys_symlinkat"
    syscall: true
    args:
    - index: 0
      type: "string"
    - index: 1
      type: "int"
    - index: 2
      type: "string"
```

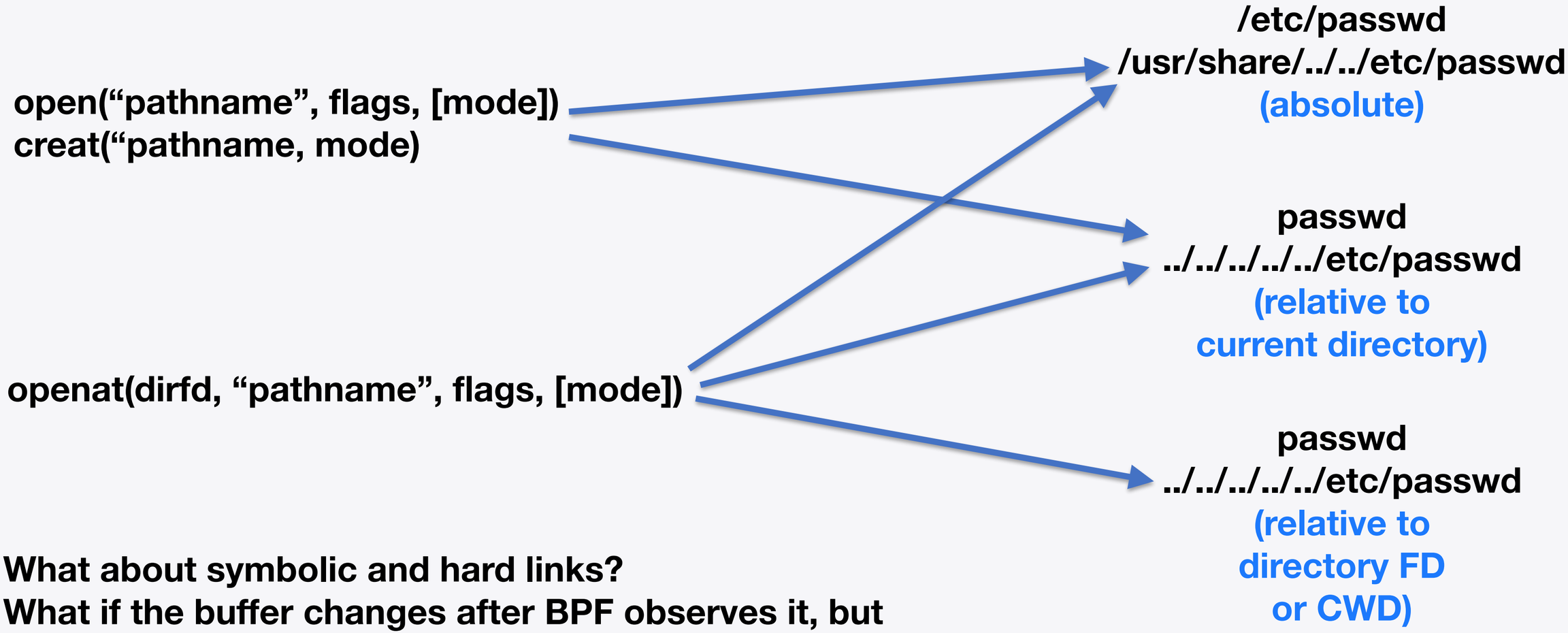
```
selectors:
- matchArgs:
  - index: 0
    operator: "Postfix"
    values:
    - "passwd\0"
  matchActions:
  - action: Override
    argError: -1
  - action: Sigkill
```

ISOVALENT

File issues

ISOVALENT

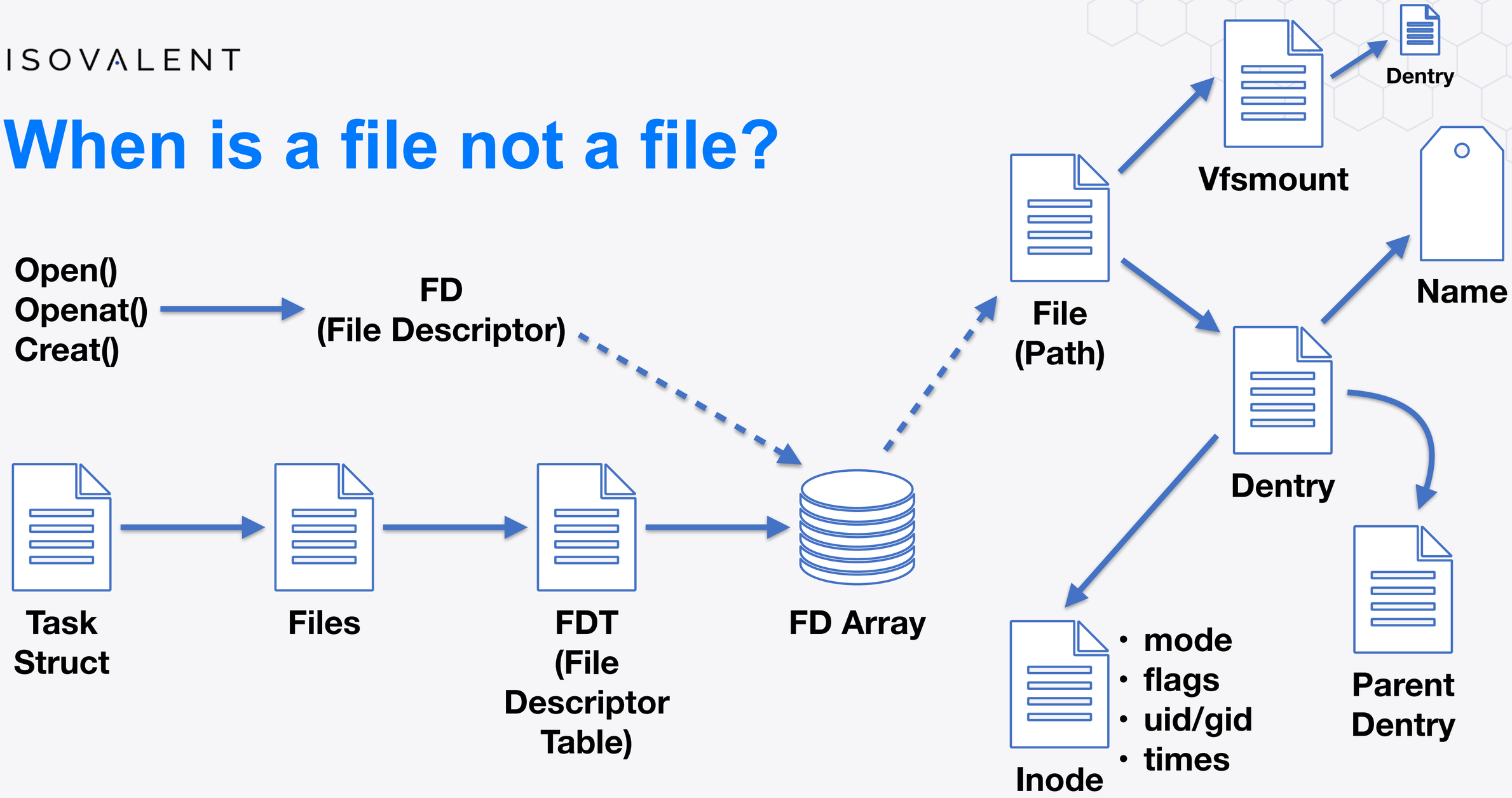
When is a file not a file?



- What about symbolic and hard links?
- What if the buffer changes after BPF observes it, but before kernel copies it?

ISOVALENT

When is a file not a file?



ISOVALENT

Follow file descriptor

- void fd_install(unsigned int fd, struct file *file)
- int close(int fd);

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "syswritefollowfdpsswd"
spec:
  kprobes:
  - call: "fd_install"
    syscall: false
    args:
    - index: 0
      type: int
    - index: 1
      type: "file"
    selectors:
    - matchArgs:
      - index: 1
        operator: "Equal"
        values:
        - "/etc/passwd"
      matchActions:
      - action: FollowFD
        argFd: 0
        argName: 1

  - call: "sys_close"
    syscall: true
    args:
    - index: 0
      type: "int"
    selectors:
    - matchActions:
      - action: UnfollowFD
        argFd: 0
        argName: 0
```

ISOVALENT

Block write to /etc/passwd

- ssize_t write(int fd, const void *buf, size_t count);

```
- call: "sys_write"
  syscall: true
  args:
  - index: 0
    type: "fd"
  - index: 1
    type: "char_buf"
    sizeArgIndex: 3
  - index: 2
    type: "size_t"
```

```
selectors:
- matchArgs:
  - index: 0
    operator: "Equal"
    values:
    - "/etc/passwd"
  matchActions:
 - action: Override
    argError: -1
 - action: Sigkill
```

ISOVALENT

Socket woes

ISOVALENT

Which process handles a socket?

- At the application / syscall layer, socket operations happen in the context of the process – synchronous
- At the network layer, it's asynchronous for lots of good reasons
- Network operations often not in context of the process that caused them / is linked to them
- So how do you link the network layer events to the right processes?

ISOVALENT

Which process handles a socket?

- sk_alloc is called by the socket syscall, returns pointer to new struct sock
- Most socket actions (connect, listen, send, sendto, recv, recvfrom) act on one of these sockets
- Track the socket and replace process details with those responsible for the socket
- accept, however, copies the listening socket with tcp_create_openreq_child and returns the new socket
- Track that too, after matching the listen socket argument

ISOVALENT

Socket tracking

- struct sock *sk_alloc(struct net *net, int family, gfp_t priority, struct proto *prot, int kern)
- static void __sk_free(struct sock *sk)

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "socket-tracking"
spec:
  kprobes:
  - call: "sk_alloc"
    syscall: false
    return: true
    args:
    - index: 1
      type: int
      label: "family"
    returnArg:
      type: sock
    returnArgAction: TrackSock
    selectors:
    - matchArgs:
      - index: 1
        operator: "Equal"
        values:
        - 2

  - call: "__sk_free"
    syscall: false
    args:
    - index: 0
      type: sock
    selectors:
    - matchArgs:
      - index: 0
        operator: “Family"
        values:
        - 2
    - matchActions:
      - action: UntrackSock
        argSock: 0
```

ISOVALENT

TCP connects

- int tcp_connect(struct sock *sk)
- void tcp_close(struct sock *sk, long timeout)

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
  name: "connect"
spec:
  kprobes:
  - call: "tcp_connect"
    syscall: false
    args:
    - index: 0
      type: "sock"
    selectors:
     - matchArgs:
       - index: 0
         operator: "DAddr"
         values:
         - "127.0.0.1/32"
```

```
- call: "tcp_close"
  syscall: false
  args:
  - index: 0
    type: "sock"
  selectors:
  - matchArgs:
    - index: 0
      operator: "DAddr"
      values:
      - "127.0.0.1/32"
```

ISOVALENT

TCP listen

- int inet_csk_listen_start(struct sock *sk)

```
- call: "inet_csk_listen_start"
  syscall: false
  args:
  - index: 0
    type: "sock"
```

ISOVALENT

TCP accept

- void tcp_set_state(struct sock *sk, int state)
- struct sock *tcp_create_openreq_child(const struct sock *sk, struct request_sock *req,
  struct sk_buff *skb)

```
- call: "tcp_set_state"
  syscall: false
  args:
  - index: 0
    type: "sock"
  - index: 1
    type: "int"
    label: "state"
  selectors:
  - matchArgs:
    - index: 0
      operator: "state"
      values:
      - “TCP_SYN_RECV"
    - index: 1
      operator: "Equal"
      values:
      - 1

- call: "tcp_create_openreq_child"
  syscall: false
  return: true
  args:
  - index: 0
    type: "sock"
  returnArg:
    type: sock
  returnArgAction: TrackSock
```

```
// include/net/tcp_states.h
enum {
        TCP_ESTABLISHED = 1,
        TCP_SYN_SENT,
        TCP_SYN_RECV,
        TCP_FIN_WAIT1,
        TCP_FIN_WAIT2,
        TCP_TIME_WAIT,
        TCP_CLOSE,
        TCP_CLOSE_WAIT,
        TCP_LAST_ACK,
        TCP_LISTEN,
        TCP_CLOSING
};
```

ISOVALENT

But UDP doesn’t have connections…?

- Just socket, send, and recv?
- What about sendto and recvfrom?
- Can send/recv UDP datagrams with tuples that don’t match the socket
- => sk_alloc isn’t sufficient
- Need to track every datagram
- Just report new tuples not seen in *n* seconds/minutes/hours

ISOVALENT

UDP datagrams

- int __cgroup_bpf_run_filter_skb(struct sock *sk, struct sk_buff *skb,
  enum cgroup_bpf_attach_type atype)

```
- call: "__cgroup_bpf_run_filter_skb"
  syscall: false
  args:
    - index: 0
      type: sock
    - index: 1
      type: skb
    - index: 2
      type: int
      label: "send"
  selectors:
    - matchArgs:
      - index: 1
        operator: "DAddr"
        values:
        - "127.0.0.1/32"
      - index: 1
        operator: "Protocol"
        values:
        - "IPPROTO_UDP"
  matchActions:
  - action: Post
    rateLimit: 30m
```

ISOVALENT

Where are we heading?

ISOVALENT

A look to the near future

- IPv6 support for sock and skb
- Improved file operations tracing
- User-friendly tracing policies and events
- More examples and documentation
- Repository of tracing policies
- v1.0

ISOVALENT

In summary

- Logging is dead; long live logging!
- BPF is amazing but still tricky
- Tetragon offers the power of BPF via the ease of YAML
- Detect, Alert, Block and Kill
- Built-in support for *file* and *network* observation
- We've still got some way to go
- And we're super committed to getting there!

ISOVALENT

Tetragon

Network & Security Observability &
Runtime Enforcement

Open Source at GitHub/Cilium/Tetragon

kevin.sheldrake@isovalent.com